Scala for Professionals

Implicits & The Type System

Type Classes

A type class provides an interface for performing certain operations on various types.

Inheritance often serves a similar role.

- Type Classes can be applied to *every* type, including final classes.
- They introduce less coupling and more flexibility than inheritance.
- Scala has no language-level support for type classes, but we can use implicits to implement them very flexibly.

Type Classes

A **Type Class** is a polymorphic trait declaring an interface:

```
trait Bool[A] {
  def isValid(a: A): Boolean
}
```

A **Type Class *instance*** is an implicit value which defines the interface for a given type:

```
implicit val intBool = new Bool[Int] {
  override def isValid(n: Int): Boolean =
    n != 0
}
```

Type Classes as Glue

A Type Class can be used as a standalone tool:

```
intBool isValid 0
// res0: Boolean = false
```

But by using an implicit parameter expecting a Type Class instance, supported types can be extended:

```
implicit class BoolOps[A](val a: A) extends AnyVal {
  def isValid(implicit instance: Bool[A]): Boolean =
    instance isValid a
}

0.isValid
// res0: Boolean = false
```

Exercise #14: Type Classes

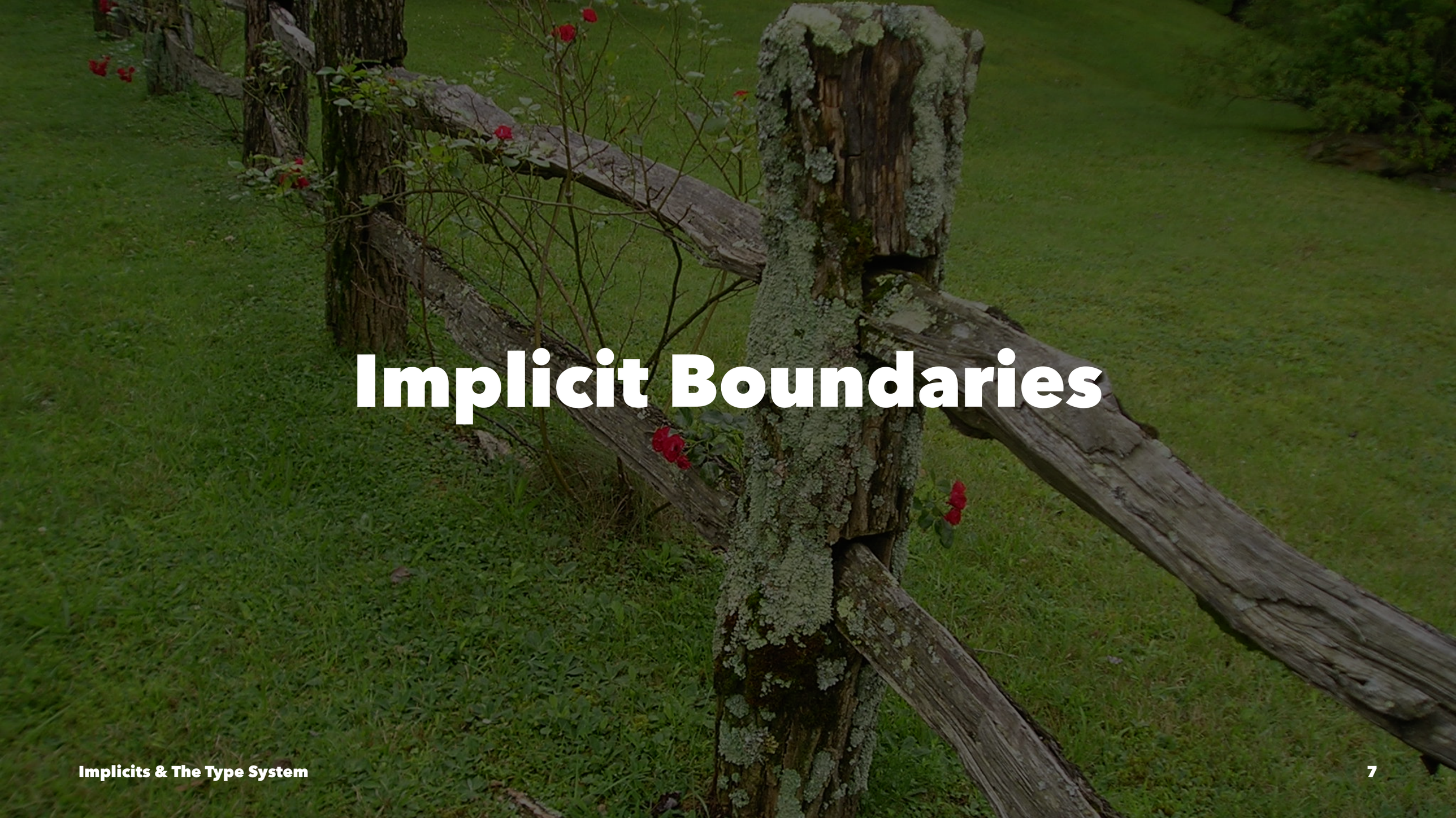
- Apply the type class pattern to create a `===` operator:
  - Create the `Equal[A]` type class declaring an `equal(a1: A, a2: A): Boolean` method
  - Add a type class instance for `Int`
  - Add a type class instance for `String`

```
1 === 1 // true
"a" === "b" // false
```

Digression: Type Constructors

- **Types** abstract over values:
  - `Int` is a type with values `..., 0, 1, 2, ...`
  - `Int => Int` is a type with function values
- **Kinds** abstract over types:
  - `*`: is the kind of instantiable data types like `Int` or `String`.
  - `* ⇒ *`: is the kind of unary type constructors, e.g. `List`.
  - `(* , *) ⇒ *`: is the kind of binary type constructors, e.g. `Tuple2`.

Parametrized types, without their type arguments, are type constructors.
- Given type arguments, they create new data types.
- Example: Applying `Int` to `List[A]` creates a `List[Int]`.



Implicit Boundaries

Context Bounds

A **Context Bound** expresses a “has a” relation to a unary type constructor. In order to define a Context Bound for a type parameter, we use the : modifier:

```
def max[A : Ordering](a1: A, a2: A): A
```

This is syntactic sugar for using an implicit parameter*.

```
def max[A](a1: A, a2: A)(implicit ev: Ordering[A]): A
```

- **Quiz**: What problem do Context Bounds potentially introduce?

---

* ev is short for ‘evidence’

`implicitly`

In order to access an implicit value by its type[†], we use the `implicitly` method:

```
implicitly[String]
// <console>:11: error: could not find implicit value for parameter e: String

implicit val s = "I am implicit"
// s: String = "I am implicit"

implicitly[String]
// res0: String = "I am implicit"
```

- **Quiz**: Where is `implicitly` defined?

[†] The rules of ambiguity & precedence ensuring we will only get one possible answer

`implicitly` and Context Bounds

We can also use the `implicitly` method to access the value of an "anonymous" implicit parameter provided by a Context Bound:

```
def max[A : Ordering](a1: A, a2: A): A =
  implicitly[Ordering[A]].max(a1, a2)

max(1, 2)
// res0: Int = 2

max(Some(1), Some(2))
// <console>:13: error: No implicit Ordering defined for Some[Int].
```

View Bounds

There's one more kind of type bound in Scala to discuss: **View Bounds**.

- A View Bound declares a “can be viewed as” relationship; to define one, we use the <% modifier:

```
case class Cage[A <% Animal](animal: A)
```

View Bounds

Usage

The View Bound is a modified kind of Upper Bound, requiring the type argument to be either a subtype of the View Bound, or having an implicit conversion to it in scope:

```
Cage("Animal")
// <console>:11: error: No implicit view available from String => Animal.

implicit def toAnimal(s: String): Animal = new Animal
// toAnimal: (s: String)Animal

Cage("Animal")
// res1: Cage[String] = Cage(Animal)
```

View Bounds vs. Context Bounds

View Bounds are discussed somewhat controversially; they may be deprecated in the future. The good news is that they can be easily replaced with Context Bounds:

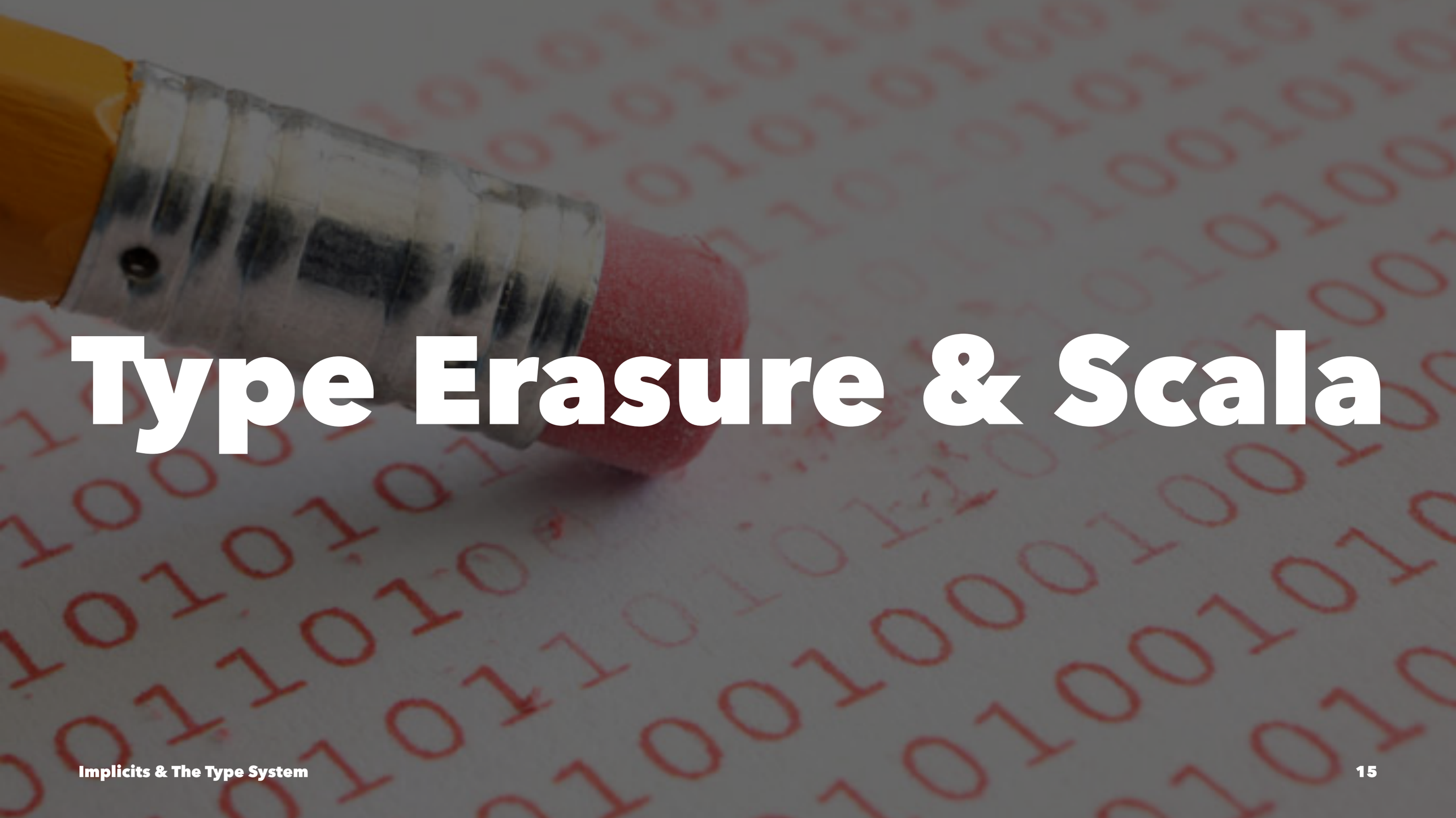
```
type AnimalView[A] = A => Animal
case class Cage[A : AnimalView](animal: A)
```

View Bounds can be replaced with Type Classes, using Context Bounds. This avoids having to implicitly wrap some object in order to extend its API:

```
def isIncreasing[A : Ordering](as: Seq[A]): Boolean
```

Exercise #15: View/Context Bounds

- There is no reason, why the `isIncreasing*` method should be restricted to `Ordered` elements
- First let's use view bounds and understand why we can call `isIncreasing*` with a `Seq` of `Int` now
- Then let's mimic view bounds with context bounds
- Finally let's use context bounds and type classes instead



Type Erasure & Scala

Tricking Type Erasure using `ClassTag`

Because the JVM has 'Type Erasure', type arguments are *never* available at runtime:‡

```
def conforms[A](any: Any): Boolean =
  any.isInstanceOf[A]
/*
<console>:8: warning: abstract type A in type A is unchecked
since it is eliminated by erasure
*/
```

‡ Type Erasure uses the type argument at compile time only – losing its information at runtime.

Tricking Type Erasure using `ClassTag`

Scala provides some compiler tricks to work around this. We can utilize a `ClassTag` Context Bound to preserve the type information:

```
def conforms[A : ClassTag](any: Any): Boolean
```

- A `ClassTag[A]` wraps a `Class[A]` runtime class.
- An implicit `ClassTag` is created by the compiler as needed.

Using `ClassTag`

To access an implicit `ClassTag`, we use the `classTag` method. The wrapped runtime class-`Class[A]`-can be accessed via the `runtimeClass` method.

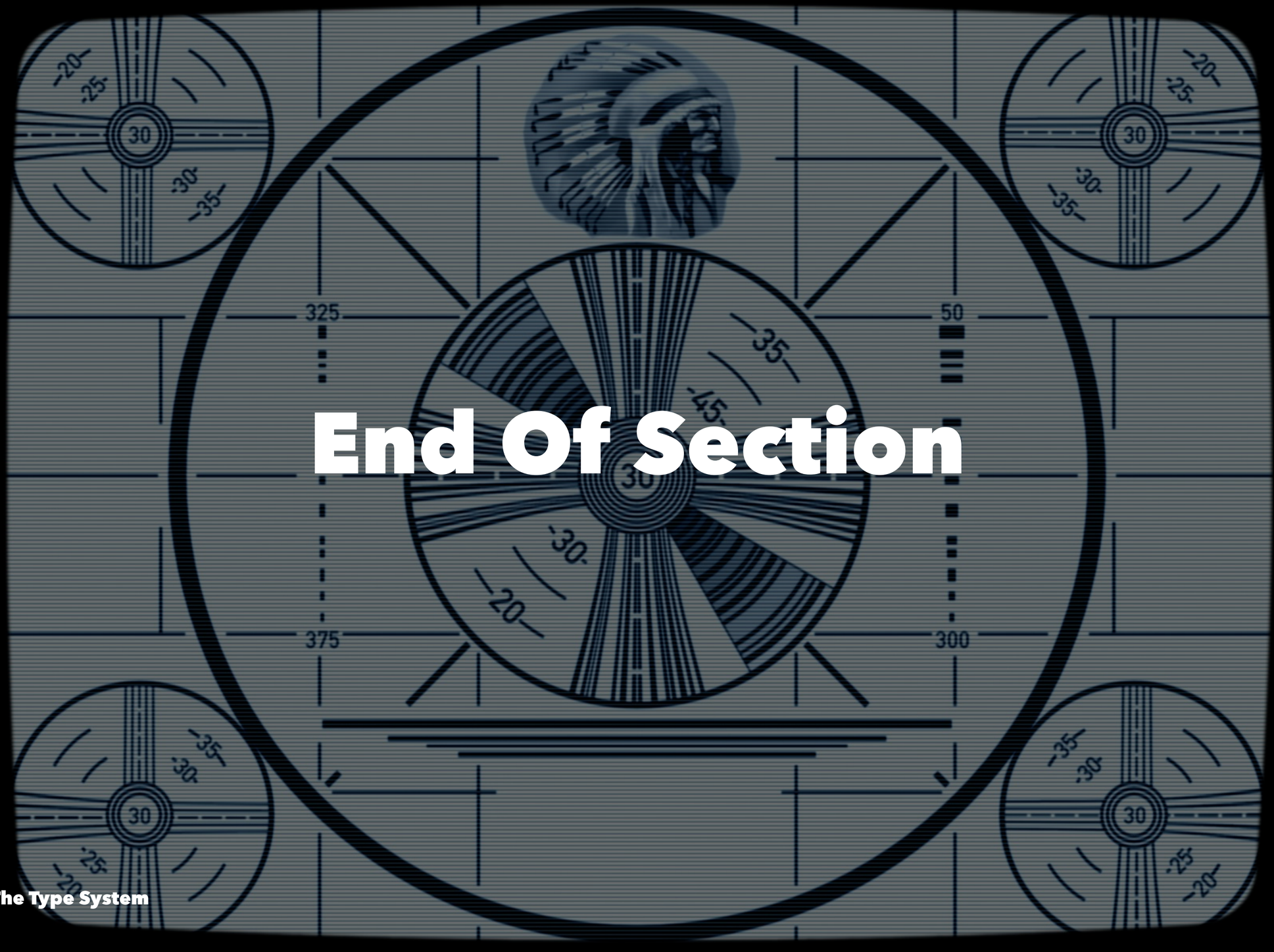
A `ClassTag` is also an extractor, i.e. it defines an `unapply` method which is used in pattern matching:

```
import scala.reflect._

def conforms[A : ClassTag](any: Any): Boolean =
  (classTag[A] unapply any).isDefined

conforms[java.util.Date](new java.sql.Date(0))
// res0: Boolean = true

conforms[java.util.Date]("")
// res1: Boolean = false
```


End Of Section